INT. J. ELECTRONICS, 1997, VOL. 83, NO. 4, 493–518

The Architecture of a highly reconfigurable RISC dataflow array processor
Sadiq M. Sait and Aamir A. Farooqui ; Int. J. Electronics, 1997, VOL. 83,No.4, 493-518

The architecture of a highly reconfigurable RISC dataflow array processor

SADIQ M. SAIT† and AAMIR A. FAROOQUI†

The architectural design and VLSI implementation of a highly reconfigurable dataflow RISC (DF-RISC) processing element (PE) are presented. This processor forms an element of a processor array (DF-RISC-PA) which possesses the features of both static and dynamic dataflow models. The array can be programmed to execute arbitrary algorithms (recursive or irregular) in both static and dynamic manner. The processor array is modelled at the behavioural level in VHDL. The gate level implementation and VLSI layout of both the PE and the array are obtained with the help of OASIS Silicon compiler by translating the VHDL model to Logic3. Sample computations are mapped to illustrate functionality. The design is validated at all levels of abstraction. The results of simulation of the PE array are presented. The architecture is compared with previous approaches. The prototype PE requires 4261 CMOS gates and uses an area of $7512 \times 8081\ \mu m^2$.

1. Introduction

Digital signal processing, image processing, real-time processing and pattern recognition require a variety of mathematical and algorithmic computations. The algorithms used for these computations possess properties such as regularity, recursiveness and locality, which can be exploited for the efficient implementation of arrays of processors. The most popular array processors are systolic arrays and wavefront arrays. These processors are effective in solving highly regular problems such as array multiplication and array addition. They are also called algorithm-oriented processors. However, when these processors are used to compute algorithms that are not highly regular, their performance degrades; moreover, they are also overburdened with complex programming problems.

One way to implement irregular computations is to use a dataflow processor array. In dataflow processors, at any given time, different PEs (processing elements) execute different parts of the code, or wait for data (depending on the data dependencies) from other processors before performing a computation. The computation is first translated to a dataflow graph (DFG) which is then mapped onto an array of processors. Different parts of the DFG are mapped to different PEs. Mapping a DFG to PEs is a combinatorial optimization problem. The cost to be optimized may include the number of PEs used, the completion time of the program, etc.

The dataflow model of execution offers attractive properties for parallel processing. Firstly, it is asynchronous because it bases instruction execution on operand availability, synchronization of parallel activities is implicit in the dataflow model. Secondly, it is self scheduling. Except for data dependencies in the program, dataflow

Received 5 August 1996; accepted 3 February 1997.

†King Fahd University of Petroleum and Minerals, KFUPM Box 673, Dhahran-31261, Saudi Arabia. Tel: +966 (3) 860 2217; Fax: +966 (3) 860 3059: e-mail: sadiq@ccse.kfupm.edu.sa.

instructions do not constrain sequencing; hence the dataflow graph representation of a program exposes all form of parallelism, eliminating the need to explicitly manage parallel execution.

2. Previous work

Owing to its simplicity and elegance in describing parallelism and data dependencies, the dataflow execution model has been the subject of many research efforts. The pioneering work in the field of dataflow processors was carried out by Dennis (1973). His work forms the basis of the static dataflow architecture later developed into what is popularly known as the MIT dataflow architecture (Arvind and Nikhil (1989). In this implementation (Dennis *et al.* 1983) a small engineering model consisting of four micro-coded processors was built. This initial model suffered from drawbacks such as lack of support for procedure calls; non-strict operators and general recursion. Despite its drawbacks, the MIT static dataflow architecture introduced new ways of thinking in the direction of massively parallel computers.

Contrary to static dataflow, another model based on tagged tokens, known as the 'dynamic dataflow' model, was proposed (Gaudiot and Bic 1997). In this model a node is enabled as soon as tokens with identical tags become available at each of its input arcs. The MIT tagged token dataflow machine was the first developed on the above principles of tagged token. This machine consisted of a number of PEs connected through an n-cube packet switching communication network to a set of specialized storage elements. It was emulated on the multiprocessor emulation facility which consisted of 32 TI Explorer Lisp Machines interconnected by a high speed network; this has been operational since 1986.

The experience gained with the above MIT tagged token dataflow computer was used to build a prototype of its successor, the Monsoon Architecture (Papadopoulos 1988). Currently, a 64-bit single processor wire-wrap prototype of the Monsoon architecture is operational.

The same principle of tagged token dataflow was also developed independently at the University of Manchester. The resulting Manchester dataflow computer (Gurd *et al.* 1985) is the first actual hardware implementation of a dynamic dataflow computer ever built. The inadequacy of handling complex data has been perhaps the most serious drawbacks of the Manchester dataflow computer. Though it was too small to run any actual application, it was able to demonstrate that pipelines in a dataflow computer can be kept busy almost effortlessly.

The SIGMA-1 (Yua *et al.* 1984) is the most ambitious tagged token dataflow computer built to date. It is a supercomputer for large-scale numerical computations, and it has been operational since early 1988 at the Electro-Technical Laboratory in Japan. It consist of 128 PEs and 128 structure elements interconnected by 32 local networks and one global two-stage omega network. Sixteen maintenance processors are also connected with the structure elements and with a host computer for I/O, system monitoring and maintenance operations.

The tagged token architecture has also suffered from certain drawbacks, such as large amount of memory needed to store tokens, which make it impractical and inefficient. Moreover, they perform quite poorly with sequential code. This is due to the overhead associated with token matching, communication, instruction scheduling and storage access.

All the above stated dataflow architectures were designed for very complex floating point operations with complex multichip hardware designs. A low-hardware-complexity dataflow PE for eight-bit fixed point operations was proposed by Weiss *et al.* (1993). Low complexity makes possible the fabrication of a VLSI chip containing 50 to 100 PEs. Several such PEs can then be put together to form a larger processor array (Koren *et al.* 1988).

To execute arbitrary algorithms effectively, a highly reconfigurable dataflow processor array as been designed in this research. In this design all PEs are initially identical (preferable for VLSI implementation) with PE identifiers assigned during configuration. Depending on the computation implemented, the array can be configured to any desired topology. To further increase the speed and reduce the area of a VLSI chip, a RISC (reduced instruction set computer) methodology (Tanenbaum 1990) has been adopted.

The paper is organized as follows. Sections 2 and 3 contain the design goals and the proposed instruction set for the processing element (PE). The complete architectural design of the processing element is explained in §4. This section includes the design of the ALU unit, instruction memory and data memory, and finally the parallel dataflow and instruction execution control unit. The section also discusses the configuration registers, their operation and organization (§4.4), a discussion of a direct matching unit essential for dynamic dataflow execution (§4.5), and finally the hardwire control and instruction decoder. Sections 5 and 6 cover the array topology and global network controller design. In §7 the implementation details are discussed. The processor operation is explained with the help of a small example, and simulation results are presented.

2. Design goals

Many simulation studies of dataflow processor systems have been reported but only a few dataflow processors have been constructed. Examples of these include Sigma-1 (Yua *et al.* 1984) and Epsilon-2 (Grafe and Hoch 1990) which are very complex and not easy to implement on a single VLSI chip. In this paper we describe the design and implementation of a low complexity dataflow array processor. The following paragraphs elaborate the design goals.

As discussed above, the static dataflow is easier to implement. It is best suited for many numerical applications that demand the computing power of supercomputers (Gaudiot and Bic 1991). On the other hand, the tagged token architecture appears best suited to applications which require iterative constructs and re-entrancy that have a less regular structure. The tagged token architecture can efficiently control the parallel and concurrent execution of multiple functions.

The static dataflow model has performance drawbacks when dealing with iterative constructs and re-entrancy, and tagged token dataflow performs quite poorly with sequential code (Gaudiot and Bic 1997, Lee and Hurson 1994). This is due to the overhead needed with token matching, communication, instruction scheduling and structure storage access.

Our first design goal was to incorporate features of both execution models (that is, dynamic and static). In the proposed design the execution is normally static. However, when data are required by more than one PE, or when PEs are on different buses, then tagging is incorporated. This combination not only helps to increase the

processor performance, but also makes the matching unit design (required for dynamic dataflow execution) simple.

Reconfigurability is another design goal. It gives maximum flexibility to the programmer to programme arbitrary algorithms in the manner that he likes, depending on his experience and heuristic. With the help of reconfiguration the programmer can switch the system architecture to adapt to the peculiarities of algorithms. The reconfiguration capability guarantees fault tolerance in applications which require real-time, maintenance-free, uninterrupted operation for long periods of time.

As we target a specific class of applications, mainly DSP, which requires intensive computation on continuous data, we know the types of instructions that are required to be implemented. Therefore, for simple and high performance design RISC methodology can be adopted. This will not only help to reduce area, but will also provide high-speed performance. In dataflow computing (mainly dynamic dataflow) a RISC architecture is not welcome, due to overhead needed in token matching for complex operations. However, in the proposed processor, because of the static architecture it is possible to implement difficult instructions or routines using RISC architecture without performance degradation due to token matching. Based on the above requirements, the design goals of the processor are that

(*a*) it must possess the features of static and dynamic dataflow models

(*b*) it must be highly reconfigurable, so that it

  (i) can execute arbitrary algorithms

  (ii) is fault tolerant

  (iii) gives maximum flexibility to the programmer

(*c*) it must have high performance, therefore an RISC architecture must be used

(*d*) its hardware should be simple so that it can easily be implemented in VLSI

(*e*) its architecture must be modular and easily extendible to larger arrays.

3. Instruction set

One of the goals of designing the Dataflow RISC Processor Array (DF-RISC-PA) was to obtain high performance with as little complexity as possible. Therefore, before defining the instruction set the complexity and performance of each instruction was carefully evaluated using VHDL simulations and heuristics. As our applications are restricted to algebraic expressions which are computation-intensive, normally appearing in applications such as DSP, after a thorough scrutiny of 32 instructions we selected 25 for this processor array. In the selection of the instruction, we used suggestions given by Patterson (Tanenbaum 1990) such as measuring the frequency of most often occurring operations. All instructions are 16 bits long, with a similar format and a single register-to-register addressing mode.

As mentioned earlier, this processor is mainly targeted on DSP applications. Therefore, the instruction set contains mainly arithmetic and logical instructions (14 in all). The 60% arithmetic and logical instructions perform two functions at the same time: they perform the required operation, and they move the result to a new destination (see Table 1). To achieve a single machine cycle multiply and divide operations, we design a special hardware multiplier and a divider.

Figure 1 gives the various instruction formats of the processor. Instructions can be classified as follows.

Instruction	Opcode	Operands	Comments	
Add	00001	Rs_1, Rs_2, Rd	$Rd \leftarrow Rs_1 + Rs_2$	integer add
Sub	00010	Rs_1, Rs_2, Rd	$Rd \leftarrow Rs_1 - Rs_2$	integer subtract
Mult	00011	Rs_1, Rs_2, Rd	$Rd \leftarrow Rs_1 \times Rs_2$	integer multiply
Div	00100	Rs_1, Rs_2, Rd	$Rd \leftarrow Rs_1 \div Rs_1$	integer divide
Cmp	01001	Rs_1, Rs_2	$Rs_1 - Rs_2$	1's compare
Inc	01010	Rs, Rd	$Rd \leftarrow Rs + 1$	increment
Dec	01011	Rs, Rd	$Rd \leftarrow Rs - 1$	decrement
Neg	01011	Rs, Rd	$Rd \leftarrow \bar{R}s - 1$	2's compl.
AND	00101	Rs_1, Rs_2, Rd	$Rd \leftarrow Rs_1 \& Rs_2$	logical AND
OR	00110	Rs_1, Rs_2 Rd	$Rd \leftarrow Rs_1 + Rs_2$	logical OR
EXCL	00111	Rs_1, Rs_2, Rd	$Rs \leftarrow Rs_1 \oplus Rs_2$	logical EXOR
Not	01101	Rs, Rd	$Rd \leftarrow \bar{R}s$	1'compl
Shl	01110	Rs, Rd	$Rd \leftarrow Rs\|0$	shift left
Shr	01110	Rs, Rd	$Rd \leftarrow 0\|Rs$	shift right
Mov	10000	Rs, Rd	$Rd \leftarrow Rs$	move
Movz	10001	Rs, Rd	$Rd \leftarrow Rs$	move if zero
Move	10010	Rs, Rd	$Rd \leftarrow Rs$	move if equal
Movl	10011	Rs, Rd	$Rd \leftarrow Rs$	move if less
Movg	10010	Rs, Rd	$Rd \leftarrow Rs$	move if greater
Creg	11000	Rd	$Rd \leftarrow 00$	clear register
Clrc	11001	Rd	$Rd \leftarrow 00$	clear carry bit
Setc	11010	Rd	$Rd \leftarrow 01$	set carry bit
Split	10101	Rs	$Bus \leftarrow Rs$	broadcast Rs
Spltg	10110	Rs	$G - Bus \leftarrow Rs$	glob. broadc. Rs

There is no instruction for the Join statement, which is implemented by means of the Direct Matching Technique.

Table 1. RISC instruction set.

(*a*) A two operand instructions that consists of

(i) an op-code field between bit positions 13–9

(ii) destination address field bit positions 8–6

(iii) addresses of two operands (operand 1 and operand 2) from bit positions 5–3 and bit positions 2–0, respectively.

(*b*) The single operand instruction has the same format, except that the operand 2 field is empty.

(*c*) The Split instruction is also a single operand instruction with the difference that the result field is empty. During the execution of this instruction the PE ID# is catenated with the operand for broadcast. The Split instruction is explained below after discussion of the architecture design.

As can easily be observed, these instruction formats require a simple instruction decoder, an added advantage of RISC architecture for arithmetic computations such as multiplication and division. Instead of using software routine as is normally done in RISC processors, a special fast combinational hardware multiplier (Sait *et al.* 1995) and a divider have been designed, with a low area × time product and single cycle execution. As seen in Table I, the 25 instructions are grouped into the following four categories: Arithmetic, Logical, Move and Flow Control, and Split.

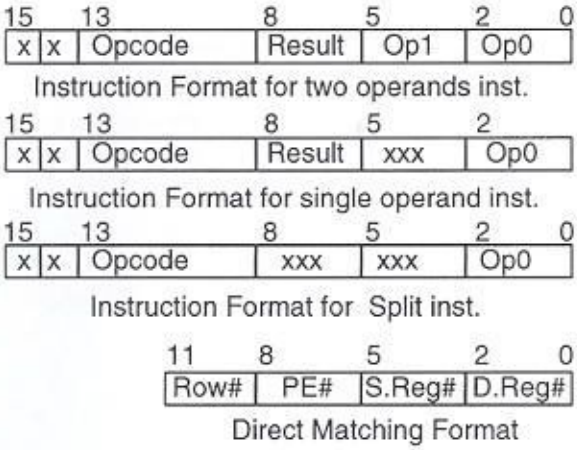

Figure 1. Instruction format of DF-RISC-PA.

The dataflow control instructions are used to control the program flow based on different conditions. These instructions are used with the compare instruction and transfer data depending on the flag setting.

4. Architecture overview

The block diagram of the PE, based on the architectural design goals of § 2 and the instruction set shown in Table 1, is shown in Fig. 2.

Each processing element forms an element of the array, and operates on eight-bit operands. All data registers are eight bits wide. Two or more instructions must be used for the execution of 16-bit and 32-bit arithmetic operands. The processing element contains the following functional blocks (see Fig. 2):

(*a*) arithmetic logic unit (ALU)

(*b*) instruction memory, data registers and FIFO RAM

(*c*) parallel dataflow and instruction execution control unit

(*d*) configuration registers

(*e*) Matching unit

(*f*) hardwired control and instruction decoder

In the following paragraphs we explain the details of the functional blocks in the PE.

4.1. *Arithmetic logic unit*

The arithmetic logic unit performs all the arithmetic and logical operations on eight-bit operands (see Table 1). In this RISC dataflow array processor, special attention has been given to the design of a fast multiplier and divider, because multiplication and division are the most frequently used functions in DSP applications. A multiplier has been designed to accomplish single cycle execution. It is based on a new algorithm reported earlier by Sait *et al.* (1995). The multiplier has been

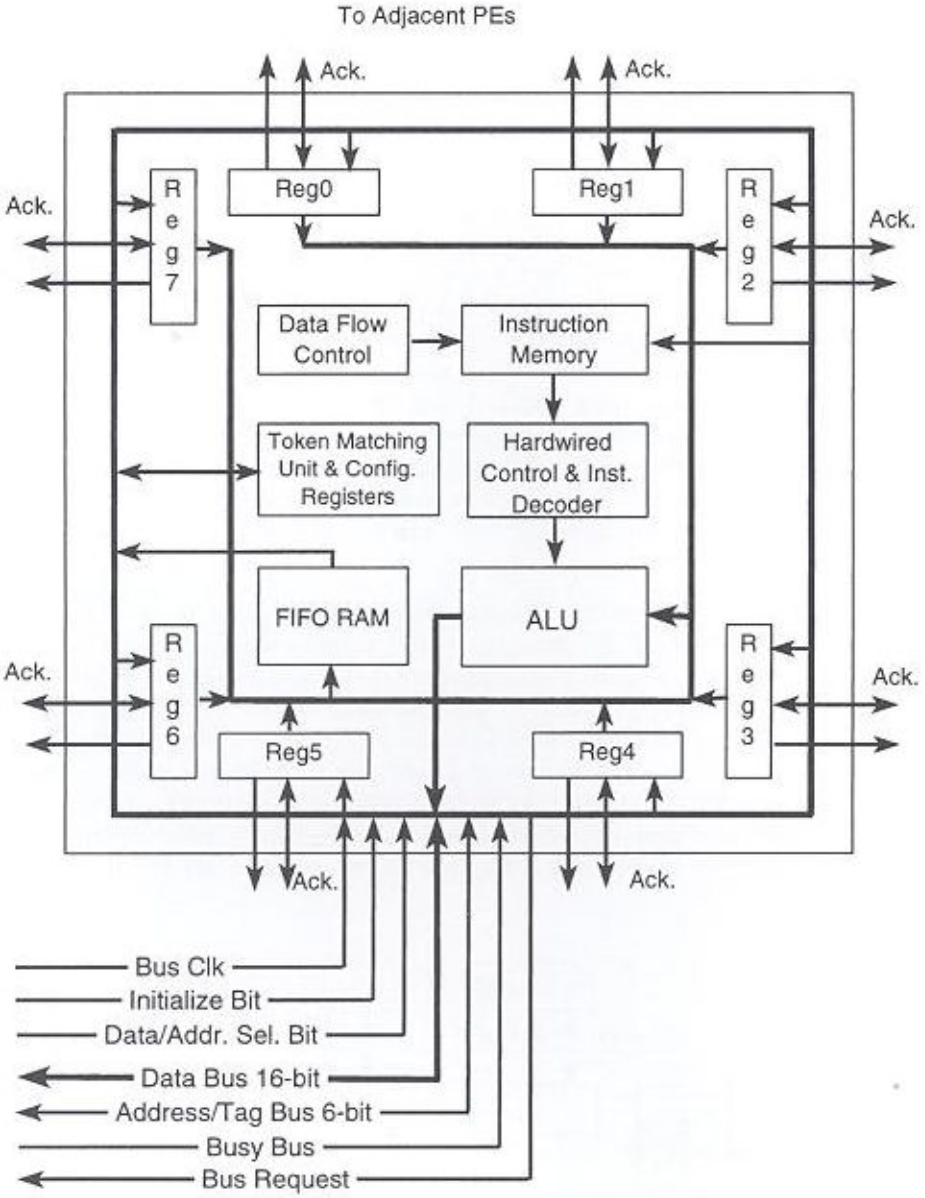

Figure 2. Block dagram of a processing element.

fabricated and successfully tested. The divider is based on the non-restoring division algorithm. It is similar to the array design proposed by Guild (1970), Majithia (1970) and Cappa and Hamacher (1973), but with some modification to achieve a higher operational speed. However, to obtain high speed, instead of using single clock multiplication (and division) these operations can also be implemented using the other instructions of the IS, thereby being consistent with the RISC philosophy.

4.2. *Instruction memory, data memory and FIFO RAM*

In this section we give details of the memory component of the processing element. Each PE has its own local instruction and data memory in addition to a FIFO RAM. The organization and working of these are explained below.

4.2.1. *Instruction memory*. The instruction memory of each processing element can hold up to eight instructions. As the instruction length is 16 bits, the size of this memory is 8×16 bits. The instructions specify the operations that each processing element needs to perform on its operands. Therefore each PE can execute a small portion of a dataflow graph independently and concurrently with other processors. These instructions are loaded from the host computer during the initialization phase and remain unchanged throughout the program execution.

Each instruction has a flag attached to it which shows its status. The flag is set when an instruction needs to be executed and is reset after the execution of the instruction. These flags together with the priority encoder help to execute an instruction in a dataflow manner. If the flag is reset then the priority encoder removes that instruction from the execution list, and selects instructions for execution whose Execution flags are set.

4.2.2. *Data memory*. Each PE also has eight eight-bit registers for data storage which lie on its periphery. These registers have a dual purpose. First, they are used to hold the operands needed in the execution of the instruction and the results obtained after execution. Second, they are also used to communicate data between neighbouring PEs (static dataflow program execution) as shown in Fig. 3. Each PE is connected to its eight immediate neighbours using these registers. The data transfer between PEs via these registers takes place in a single clock cycle.

An eight-bit configuration register called a DTCR (data transfer control register) is used to control transfer of data between adjacent PEs. Each bit of the DTCR

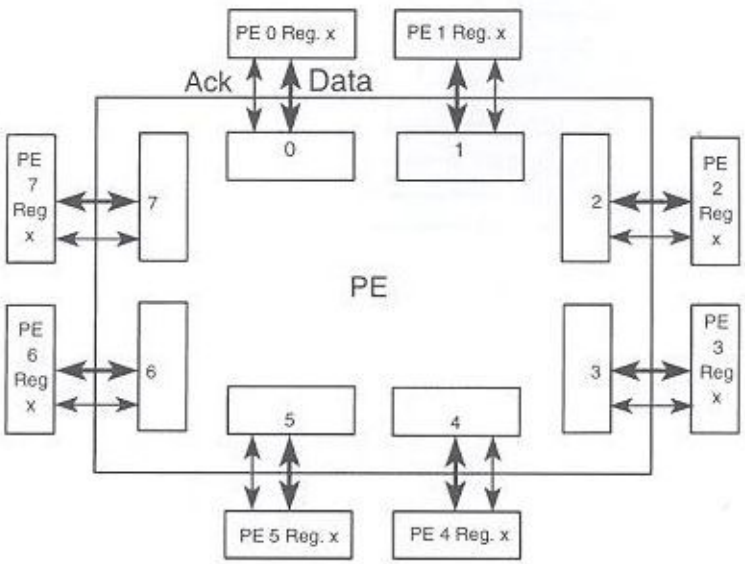

Figure 3. Data register interconnections with the adjacent PEs registers.

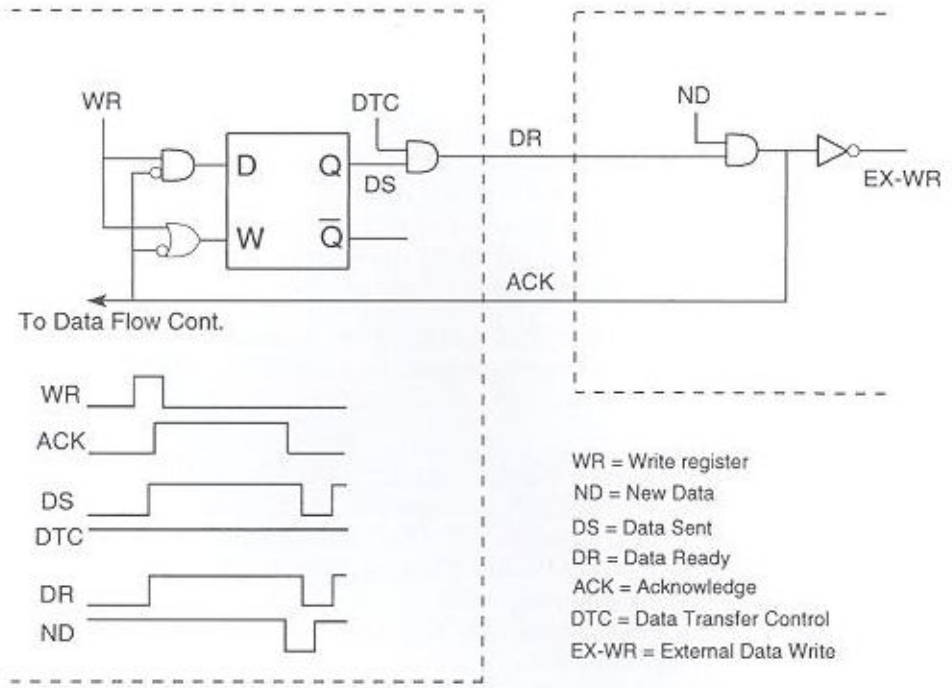

Figure 4. The proposed handshaking circuits and its timing diagram.

corresponds to one of the data registers of the PE. The physical connections between adjacent processing element registers exist all the time. When a particular bit in the DTCR is set, only then is the transfer enabled. The transfer between PEs is dependent on the mapping of the dataflow graph onto the processor array.

Associated with each data register is a flag called the New_Data flag. The bit value of this flag indicates if new data have arrived or can be received from the adjacent PE. The New_Data bit is set when a new data is written into the register, and is reset after the consumption of data. The New_Data bits are applied to the input of the parallel dataflow and instruction execution control unit (PICU) which controls the program execution (explained in §4.3).

To ensure correct sequencing and data transfer between adjacent PEs a handshake protocol must be adopted to synchronize the operations. There are two types of asynchronous scheme (Kung *et al.* 1987), the one-way control scheme and the two-way control scheme. In the one-way control scheme the sender sends data without waiting for the acknowledgment signal of the receiver. This method is suitable for an array processor only when large buffers are provided at the receiver. The two-way control scheme usually known as handshaking is preferable for most processors arrays. The two-way communication scheme is adopted in this work. The proposed handshaking circuit designed in this work is illustrated in Fig. 4. It uses a flip–flop and four gates, and operates as follows.

Step 1. Initially all ACK signals are 0. When data are written into a register (WR = 1) then DS is set on the second phase of the system clock.

Step 2. If a Data_Transfer_Control bit (in the DTCR) corresponding to the same register is set, then a Data_Ready (DR) signal is produced for the adjacent processing element.

Step 3. In the adjacent PE, the DR signal is ANDed with the New_Data bit of the corresponding register of adjacent PE. If New_Data bit is set (the register contains a valid data) then a high acknowledge signal (ACK) is transmitted to the data transmitting PE, indicating that the data is not accepted. Owing to this acknowledge signal the PICU does not execute those instructions which write on the data transmitting register (static dataflow execution).

Step 4. As soon as the New_Data bit of a data register is reset, data are written into that register and the acknowledge signal becomes low, indicating that data have been transmitted properly.

4.2.3. *FIFO RAM*. To transfer data to more than one PE a Split instruction is used (for dynamic dataflow execution). This instruction broadcasts the data on the Host/PE data bus with a tag attached to it. The tag is formed by catenating the Source Register number and the PE identification number (PE ID#). The processing elements which need this data, accept it.

As the traffic on the bus is higher than anywhere else in the processor array, a FIFO RAM is used to hold the data before they are transmitted on the bus. The Split instruction transfers the data to the FIFO RAM in a single cycle, and then in subsequent cycles the data are transferred to the bus. This can be done in parallel with the normal program execution. As soon as the FIFO RAM receives new data it sends a Bus_Request to the Global Network Controller (GNC) (discussed later in §6), which transfers the bus control to the processing element with the highest priority. After gaining control of the bus, the processing element transfers its data to the bus in a single clock cycle, and updates the FIFO RAM in the following cycle. Each processing element has a FIFO RAM of 8×11 bits. This can be used for other purposes depending on the program. In case the RAM becomes full, the PE sends the highest priority signal to the network controller requesting the bus for immediate transfer of RAM data in the subsequent cycle.

4.3. *Parallel dataflow and instruction execution control unit (PICU)*

The PICU is the most important component of the DF-RISC-PA. The instructions in the processing element are not executed in any predetermined order. Instead, the arrival of all the operands for a certain instruction enable the execution of that instruction. The PICU controls the data-flow operation by constantly checking all the eight instructions concurrently for the availability of the operands. There are two types of flags which monitor the data movement (among registers) and instruction execution. They are, the New_Data flags (mentioned earlier) and the Execute flags.

Each data register has a New_Data flag attached to it. This flag indicates whether the register has a valid operand or can receive a new operand. If a flag corresponding to a register is set, it means that the data in that particular register is valid and can be used for instruction execution. The reverse is true when the flag is reset. This unit continuously and concurrently monitors the New_Data flags and looks for instructions that need operands for their execution.

Flags that monitor the instruction execution are called Execute flags. Each instruction has an execution flag (see Fig. 5) which resets after the execution of the instruction. These flags are used by the instruction sequencing unit for the execution of one instruction at a time. With the help of New_Data and Execute flags, ACK signals from adjacent processing elements and a data transfer control configuration register, PICU controls the instruction execution. A high Execute_Instruction signal is generated when all the following conditions are true:

(*a*) operand-0 of an instruction has arrived (for single operand instructions)

(*b*) both operands have arrived (in case of two-operand instruction)

(*c*) instruction Execute flag is 1

(*d*) the acknowledge signal, corresponding to the set bit of the DTCR, is low

(*e*) the operand code represents a valid instruction

It is possible that many instructions become ready for execution at the same time, depending on the arrival of the operands. Parallel execution of all the instructions is not possible due to resource limitations. Therefore the output of the PICU is applied to an execution circuit containing a priority encoder (see Fig. 5). This circuit enables one instruction at a time. The Execute_Instruction signals (IEX) of the dataflow control unit are ANDed with the instruction Execute flags (EX) and applied to the input of the priority encoder. This encoder selects the highest priority instruction for execution. The instruction Execute flag resets after instruction execution and is set again after all the instruction Execute flags are reset (i.e. if all the instructions are executed) or the priority encoder output is zero (i.e. if no instruction is ready for

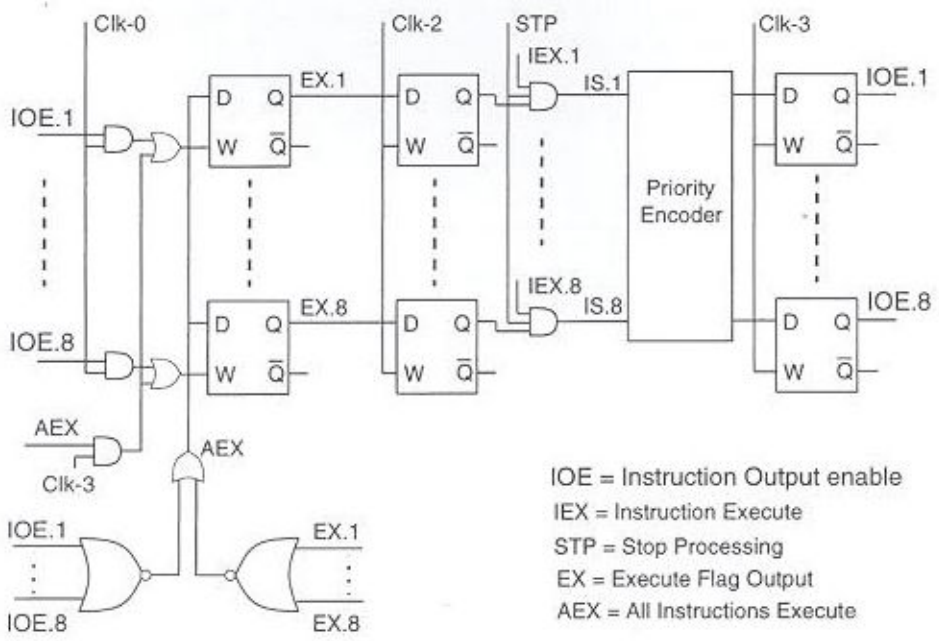

Figure 5. Execution unit logic diagram.

execution). By doing so, an instruction is not executed again (even if it has received all its operands) unless all the instructions are executed or no other instruction is ready for execution.

4.4. *Configuration registers*

The configuration registers give the reconfigurative characteristic to this processor array. There are four configuration registers (three used currently and one spare) as shown in Fig. 6. The three registers are DTCR (data transfer control register), HDRR (host data request register) and a PE_ID# register (to hold the processor identification number).

The DTCR is used to control the transfer of data between the registers of adjacent processing elements. If bit i of this register is set, then data are transferred between PE register i and its neighbouring PEs register, to which it is connected. This transfer takes place with the help of a simple handshaking protocol, without any software control (see §4.2).

The HDRR is an eight-bit register, each bit of which corresponds to one of the eight registers of the PE. The bits are set/reset during initialization. If any bit of this register is high and if the New_Data bit is low, then a signal is generated to the global network controller requesting for data. The destination of the data is the register corresponding to the high bit position of the HDRR. Once the data have been fetched, the New data bit attached to the register that received the new data is set. This technique is used to avoid the lengthy delays associated with load instructions.

The third configuration register is called the PE_ID# register. With the help of this configuration register, a processing element can be assigned any identification number. This identification number is used as a tag in dynamic dataflow execution, and in communication with the host. This feature makes the design very attractive for VLSI implementation because all the PEs are identical; therefore only one set of masks is required to fabricate any numbers of PEs. This configuration register also provides fault tolerance. On determination of the PE failure, a spare PE can be

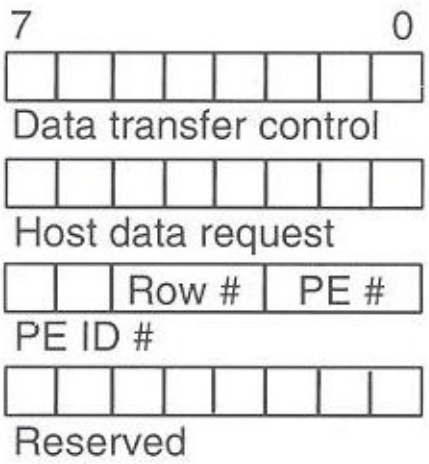

Figure 6. PE configuration registers.

loaded with failed PE_ID#, without the need to remap the program to the new set of PEs (Gaudiot *et al.* 1985).

There is also a fourth register which is still unused; this register can be used for future expansion. One possible use of this register is to hold constants (Gaudiot and Bic 1991). If bit i of this register is set, data in register i should be treated as a constant and its value should not be changed during program execution.

4.5. *Direct matching unit*

In this processor a novel and simplified process of matching of tags has been used. The direct matching technique used in this processor is slightly different from the one reported in the literature (Papadopoulos 1988, Grafe and Hoch 1990, Nikhil and Arvind 1989). It avoids the expensive and complex process of associative search used in previous dynamic dataflow architectures.

The Direct Token Matching unit is used for the dynamic dataflow execution of a program. It contains four registers of 11 bits each. In these registers the matching tags and destination register addresses are stored during the initialization phase. A bit attached to these registers indicates that the contents of the register are valid. A tag is the catenation of a PE identity number and a source register number. During runtime, whenever a processing element transmits any data on the bus (using the Split instruction as discussed in §4.2), all the processing elements on that bus compare the tag of that data with the prestored tag (see Fig. 7). If a match is found, the data are stored in a temporary memory or are transferred directly to data registers, otherwise data are discarded. By using a direct matching technique, the matching of tags is performed in only one machine cycle. Owing to the limited bandwidth of the common bus, as explained earlier, a FIFO RAM is used temporarily to hold data.

As shown in the detailed diagram of the matching unit (Fig. 8), the data along with the tag are written in a 19-bit wide latch, controlled by the GNC (discussed in detail in §6). The tag in the latch is compared with the contents of prestored tags in

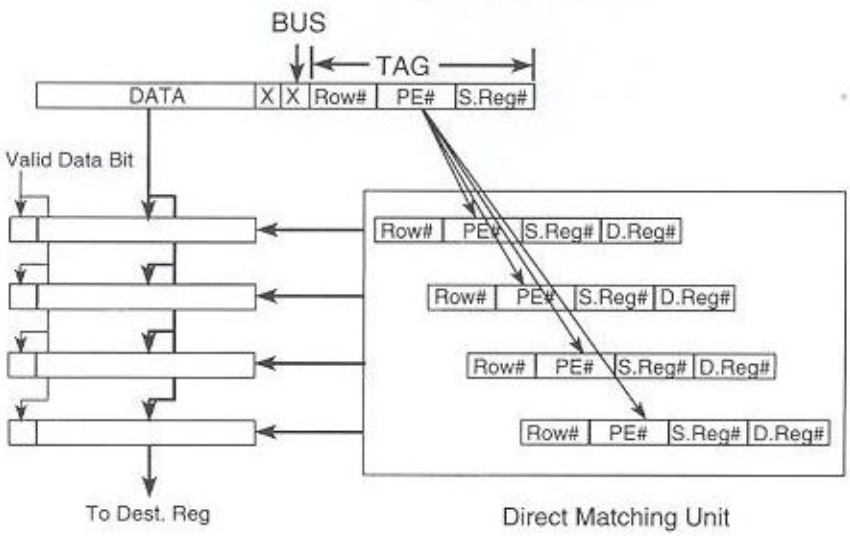

Figure 7. Block diagram of direct matching unit of the DF-RISC-PA.

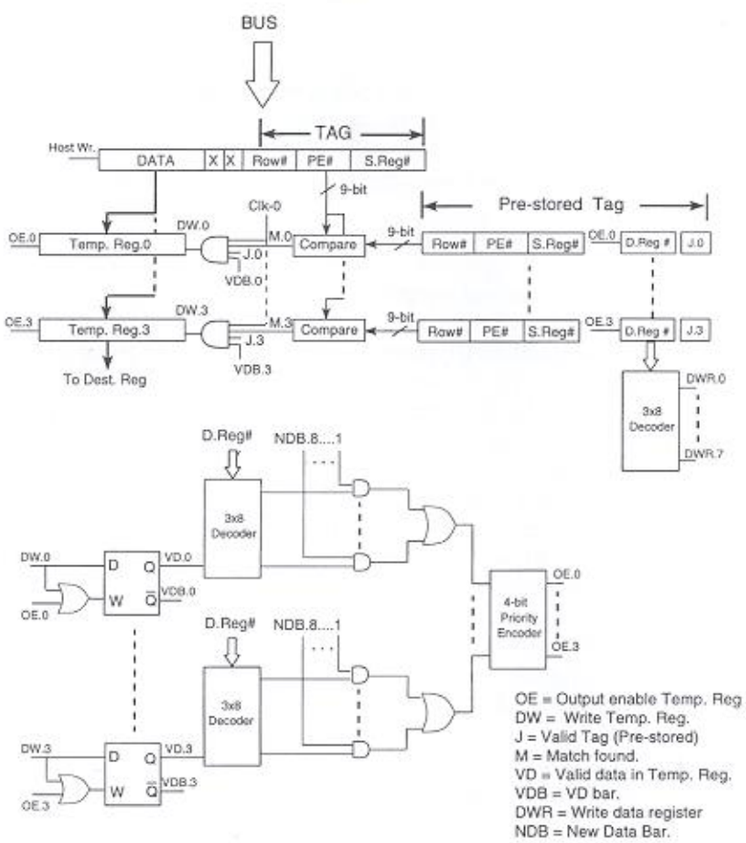

Figure 8. Direct matching unit of the DF-RISC-PA.

the matching registers. If the prestored tag is valid (indicated by the *J* bit attached to each register), and temporary register is empty (indicated by VDB) then, if a match is found, data are transferred to a temporary register. Otherwise the new data are written to the 19-bit latch using Bus_Clk controlled by GNC. The data transfer to the data registers is dependent on the New_Data bit. If this bit is zero (indicating that the register is empty), the data from the temporary register are transferred to the data registers where they are used by the instructions for program execution.

4.6. *Hardwired control and instruction decoder*

The Instruction Decode and Control unit is a simple combinational circuit containing a decoder for all the instructions. It also generates the required control signals. Each instruction is first decoded by the decoder. Then control signals are generated by the control circuitry to execute an instruction. It is the smallest unit of the processor, requiring only 87 gates for 25 instructions decoding and control.

5. Array topology

The processor array, as mentioned earlier, can be programmed to implement both regular and irregular/unstructured computations. For this purpose special attention has been given to array topology design. Based on VHDL simulations of the handshake protocol discussed earlier, a topology for processing element interconnection has been designed. Fig. 9 shows the floor-plan of the chip.

In the current prototype, 16 PEs are arranged as a square array, with four PEs per row (this can be extended to 16 PEs per row). Each PE has a six-bit unique address assigned by the programmer. To facilitate maximum communication between PEs, each PE can communicate with its eight immediate neighbours through the boundary registers/ports. It can also communicate with the other PEs and the host using the GNC and the host bus which runs between two alternate rows of PEs. Although the connections between adjacent PE registers exist all the time, the actual data transfer between them is controlled by a single bit in the DTCR, which is under the control of the programmer. There are three possible software-controlled connections between PE registers. Two of these are for the transfer of data between PEs (see Fig. 9).

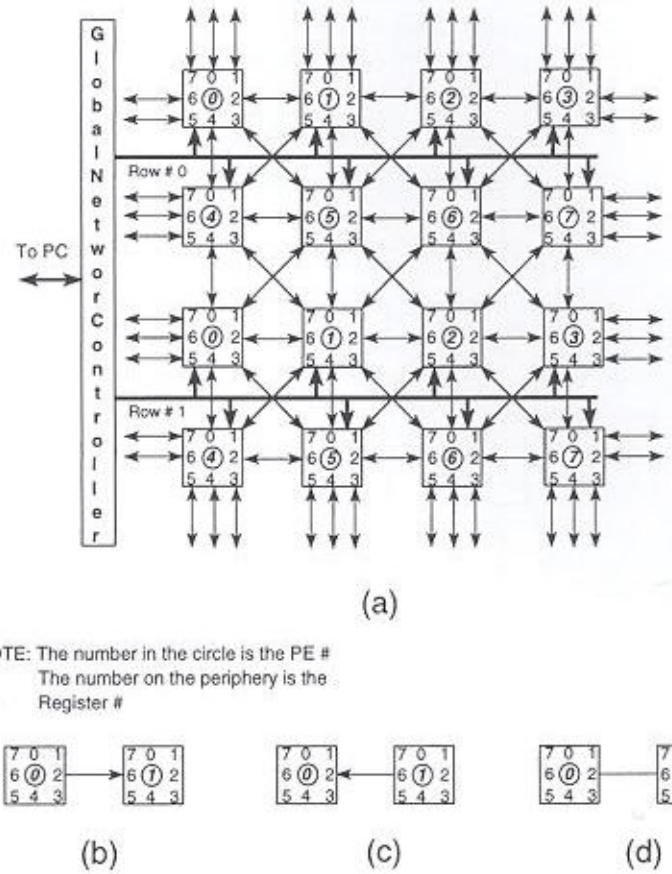

Figure 9. (a) Floor-plan of the DF-RISC-PA; (b) data transfer from PE0 to PE1; (c) data transfer from PE1 to PE0; (d) no data transfer between PE0 and PE1.

The data transfer between processing elements is controlled by a simple handshaking protocol (as discussed in §4.2). The topology shown in Fig. 9 gives maximum flexibility to the programmer and results in tighter coupling and faster communication among processing elements. Figure 9 shows the physical layout and the connections. There is no logical connectivity between the adjacent PEs until the computation is mapped onto it. This is done by software. It is possible to implement an arbitrary dataflow graph on this processor array (such as star, pyramid and binary tree.)

6. Global network controller

The global network controller (GNC) takes care of the communication between PEs and the host. It generates control signals for data transfer between the host and PEs. It receives 16 bits of input data from the host and processed data from PEs. This network controller is used to interface the processor array with the parallel port of a personal computer. Additional PEs can easily be added due to the modular nature of the communication network, and thus facilitates the future design of a massively parallel computer.

As shown in Fig. 10, the GNC contains a bus resolver and a memory management unit, a 64K byte memory, and a handshake unit. This unit has not been included in the VLSI chip of the array processor due to the following reasons.

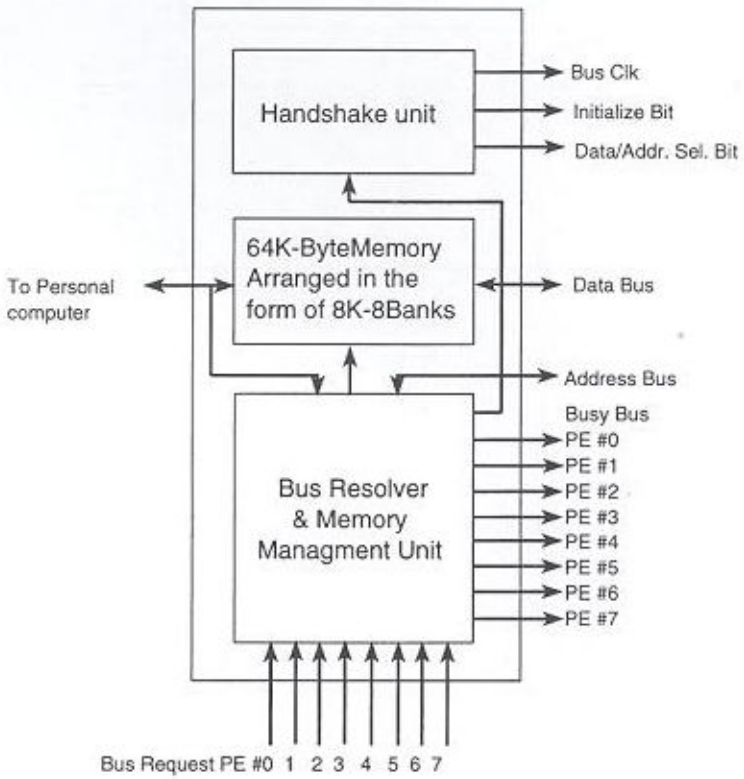

Figure 10. Global network controller block diagram.

(*a*) It requires a large amount of memory for data storage, which can be easily included in the system using off-the-shelf memory chips, instead of using the real estate of the processor chip.

(*b*) An external communication controller makes it possible to combine several processor chips for designing larger arrays.

(*c*) An external communication controller helps to probe the communication between the processing elements and network controller for debugging purposes.

(*d*) The host communication unit can be modified according to different computer systems of different vendors (at present it is designed for a 486 personal computer parallel port).

The GNC works as follows. During the initialization phase of a PE, first a PE is selected based on some priority. Following this, the GNC generates the Initialize and data/Address select signals. It then transmits these signals along with the addresses of the following over the address bus: the address for the instruction, for data register, and for matching and configuration registers. On the 16-bit data bus it sends data.

A PE requests for a data transfer, by using the Bus request signal to the GNC. Depending on the priority of the PE, the GNC transfers the control of the bus to it. Using Busy_Bus and Bus_Clk signals (only one Busy_Bus signal can be activated at a time) it writes the contents of bus into the data registers of the PEs. The GNC also writes the same data into the global network memory, which can be accessed by the host computer.

7. Implementation

This processor was first modelled in VHDL at the behavioural level. One of the many advantages of VHDL is that it supports concurrent execution of processes (Lipsett *et al.* 1987) and statements, which is very helpful in modelling a parallel processor. To produce the layout for fabrication, the VHDL behavioural model was translated to a structural level design in Logic3 (hardware description language). With the help of the OASIS silicon compiler Logic3 descriptions are translated to standard-cell VLSI layouts in 2 μm SCMOS technology (Kozminski 1992). To verify the correctness of the layout and determine the operating speed, the circuit is extracted from the layout and simulated using the switch level simulator irsim. Simulation has been conducted on both the individual PE, and the complete processor array (Mayo *et al.* 1990).

We now give an example of how a computation is mapped to the processor array.

Example 1: Consider the expression

$$x = (a + b) \times (c + d) \tag{1}$$

where $a = [a_0 \ \ldots \ a_n]$, $b = [b_0 \ \ldots \ b_n]$

$$c = \begin{bmatrix} c_0 \\ \vdots \\ c_n \end{bmatrix}, \quad d = \begin{bmatrix} d_0 \\ \vdots \\ d_n \end{bmatrix}$$

Merely for the sake of illustration, let us map this computation onto two PEs, as shown in Fig. 11. The instruction level mapping and dataflow among the registers and processing elements is shown in Fig. 12. This figure also shows how the instruction and data are stored in the instruction memory and data registers of the PEs, and the bit patterns of the configuration registers. In Fig. 12 the flow of data between PEs is shown by thick lines.

One instruction (Add) is assigned to PE0 and three instructions (Add, Mult, and Split) are assigned to PE1. The DTCR bit 2 of PE0 is set to transfer the data of PE0_Data_Reg2 to PE1_Data_Reg6. The array is simulated at both behavioural and switch level using the switch level simulator IRSIM (Mayo *et al.* 1990). The switch level model is obtained by extracting the circuit from the synthesized layout. The instruction and data memory of processing elements are loaded with the data using the command file of the simulator.

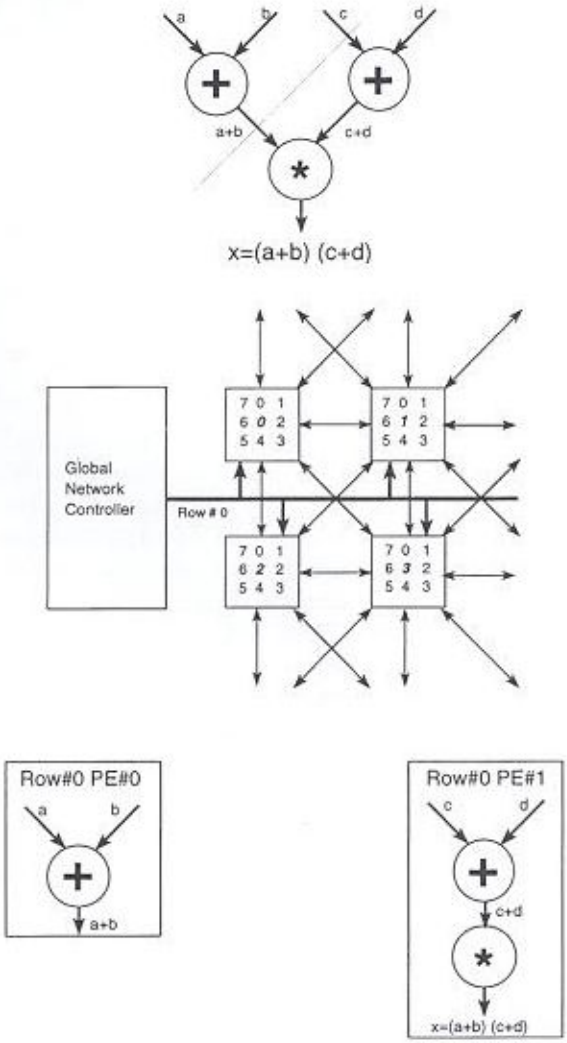

Figure 11. Mapping of the expression $x = (a + b) \times (c + d)$ on to 2 PEs.

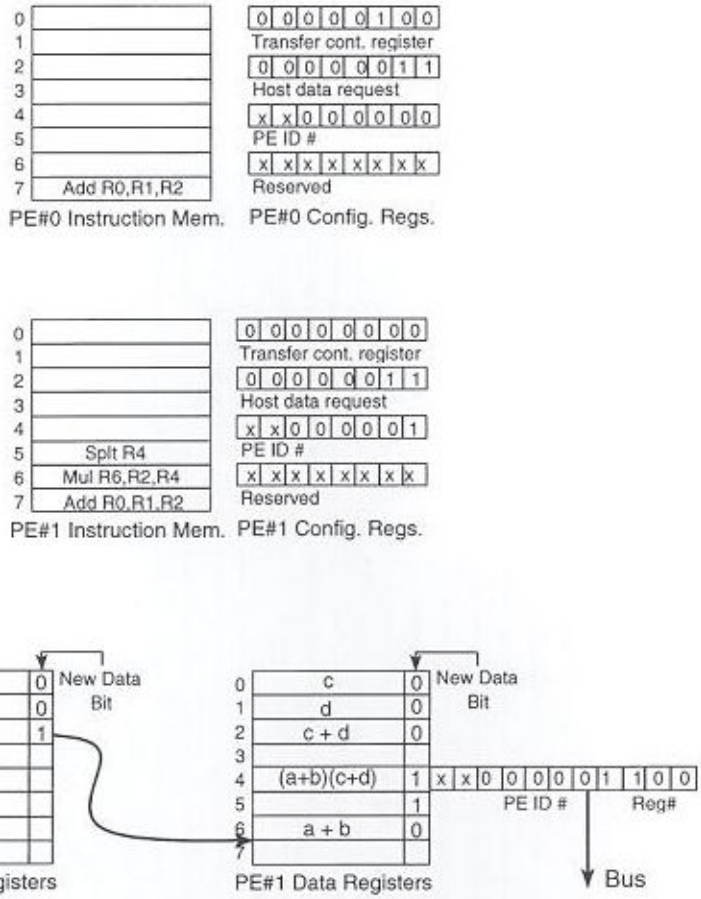

Figure 12. Instruction level mapping of the expression $x = (a + d) \times (c + d)$ on the PE array.

Figure 13 illustrates the output waveforms obtained after a simulation run. In Fig. 13 the instruction opcode and destination fields of an instruction are represented by PEx_opcode and PEx_dest vectors/signals, respectively (x denotes the PE number). The contents of data registers are shown by the PEx_Data_Reg# vectors. PEx_IOE is the instruction output enable vector of PEx. PEx_Ack_In# and PEx, Data_Ready_In# are the control signals for asynchronous communication between PEs. PEx_start and PEx_Data_Add signals are controlled by the host for PE initialization. PEx_Bus_Req (bus request) and PEx_Host_Data_Req (host data request) are the signals used for communication with GNC. PEx_Out represents the PEx output catenated to PE_ID# (tag for dynamic execution).

The clock of 500 nS (2 MHz) is used in simulations. During the initialization phase of PE0 (0 nS to approximately 10 000 nS) the instruction and data memory are initialized with the instruction code and operands. Both the registers (PE0_Data_Reg0 and PE0_Data_Reg1) of PE0 are loaded with operands in a single clock cycle using the 16-bit data bus. In the following machine cycle the PE0_IOE

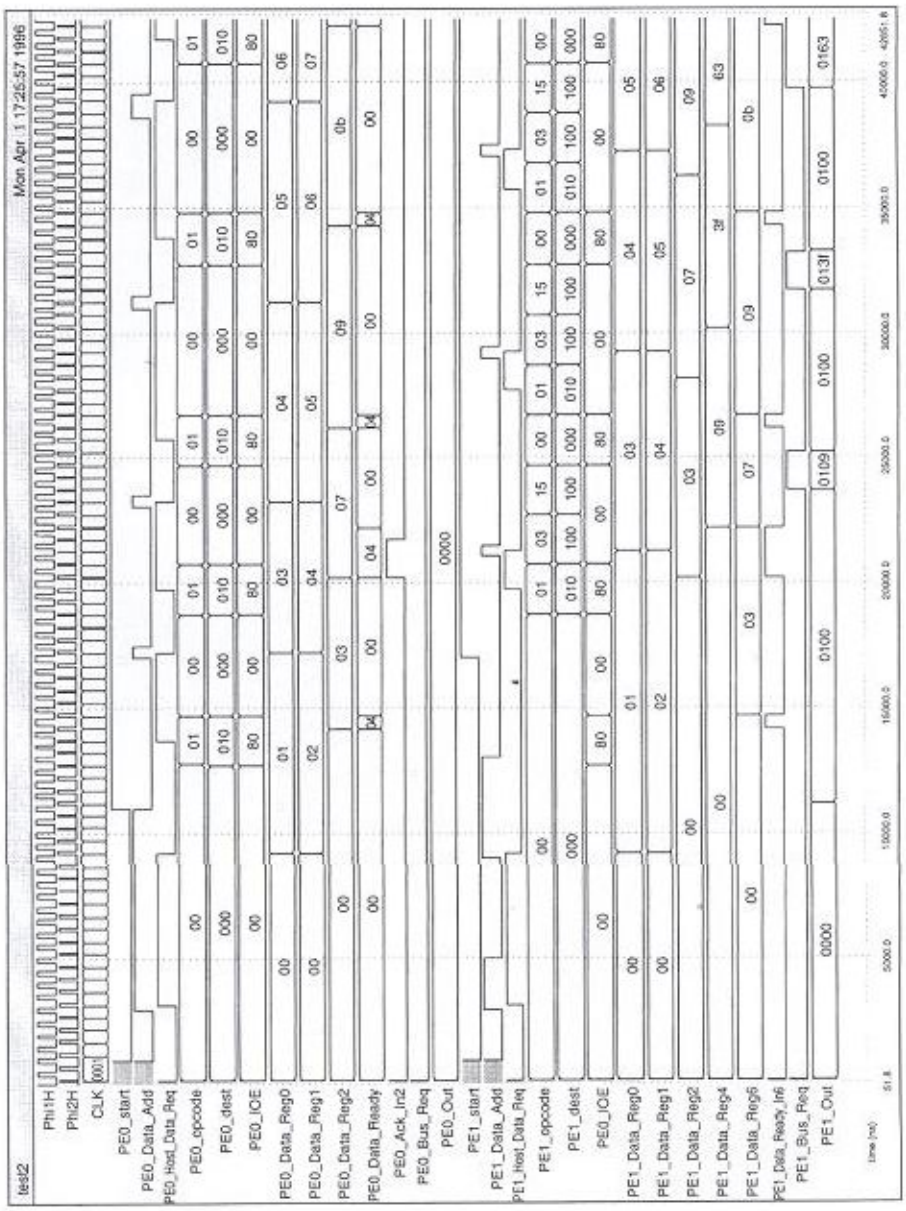

Figure 13. Switch level simulation results for vector computation $x = (a + b) \times (c + d)$.

signal for the execution of instruction (Add Reg0, Reg1, Reg2) is generated. As soon as the result is written on PE0_Data_Reg2, PE0_Data_Ready2 becomes high to send a data ready signal to PE1_Data_Reg6. This signal remains high until the data are accepted by PE1 (indicated by low PE0_Ack_In2 signal). During this period no new data can be written on PE0_Data_Reg2 (static dataflow execution).

The PE1_Data_Reg6 holds the data and calculates the final product $((a+b) \times (c+d))$ after the availability of new data in PE1_Data_Reg2 (result of $c+d$ is produced in the same way as $a+b$). Finally the Split instruction is used to transfer the result stored in PE1_Data_Reg4 to the BUS using the PE1_Bus_Req signal. The first result is produced after 24 000 nS and subsequent results are produced after a delay of 8000 nS. This reveals that the expression $x = (a+b) \times (c+d)$ has been calculated in only four machine cycles using two processing elements; this may be further reduced if proper mapping is used.

The same expression has been implemented on an 80486 personal computer. The data for a, b, c and d are stored in the same named arrays and the result is stored in the prod array. This von Neumann mroprocessor requires 32 clock cycles to execute the expression $x = (a+b) \times (c+d)$, for eight-bit operands.

Example 2: As another example, consider the computation of the expression $x + x^2 + x^4$. It is used to illustrate how the tags are matched. For the sake of explanation it is supposed that $x^2 < 256$; otherwise a 16-bit multiplication procedure needs to be used.

One of the possible mappings of the above expression on this processor array (to show dynamic and static execution at the same time) is shown in Fig. 14. The instruction level mapping and dataflow among the registers and processing elements is shown in Fig. 15. This figure shows how the instruction and data are stored in the instruction memory and data registers of the PEs. This figure also shows the bit patterns of the configuration registers and the contents of Tag Matching Registers. The flow of data from PE0 to PE1 is shown by thick lines.

PE0 executes two instructions (Mult and Split). The Mult instruction is used to calculate x^2, because the DTCR bit 5 is set, therefore the contents of PE0 Data Reg4 (x^2) are transferred to PE2_Data_Reg0 in the same clock cycle. The Split instruction is used to broadcast x^2 on the Bus by catenating the PE_ID# and source register number (tag formation) to it.

PE1 executes only one Add instruction. For this it receives one of the operands (x) from the host, whereas the other operand (x^2) is obtained by tag matching. It is clear from Fig. 15 that the tag matching units of PE1 and PE2 contain valid tags, whose values are equal to the tag broadcasted by PE0. The matching unit tags are stored during the initialization phase (they can also be changed during run time by the host), and calculated by the mapping software. The tags stored in the matching unit contain the PE_ID#, source register # and destination register #. As soon as a match is found between the bus tag and matching unit tag, the bus data (in this case x^2) is transferred to the destination register if New Data Bit is zero, or else stored in the FIFO RAM. Again the DTCR bit 5 of PE1 is set. Therefore, the addition result ($x + x^2$) is transferred to PE2_Data_Reg1.

PE2 executes two instructions (Mult and Split). The Mult instruction is used to calculate x^4, using the data stored in Reg2 and Reg0. PE2_Data_Reg0 receives the

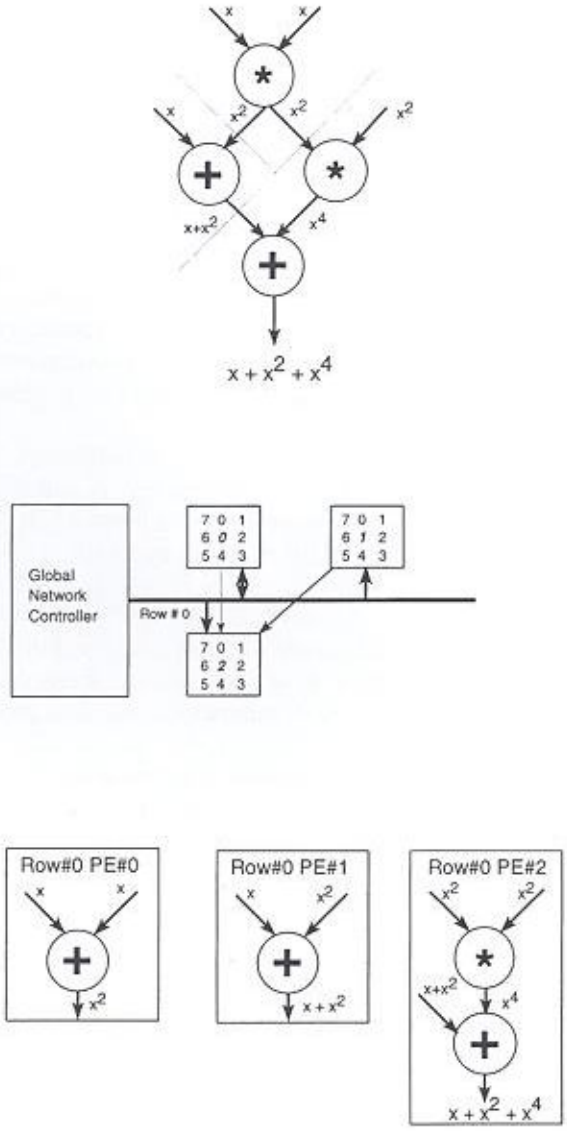

Figure 14. Mapping of the expression $x + x^2 + x^4$ on to three PEs.

data from PE0_Data_Reg4 using handshaking. The PE2_Data_Reg2 receives the data by matching the prestored tag with bus tag broadcast by PE0. Finally the Add instruction is used to calculate the sum $x + x^2 + x^4$.

8. Discussion

In this paper we present the architectural level design of a processing element. This element is used to implement an array of processors. The PE is designed using RISC philosophy and it therefore exhibits high performance and uses a small area

Instructions	25
Instruction length	16
Instruction opcode	5
Data registers	8 × 8 bits
Program memory	8 × 16 bits
Configuration register	4
PE host comm, data bus	16 bits
Transistors	*n*-channel = 15 760, *p*-channel = 14 873
Gates	4261
PE size	7512 × 8081 μm^2
PE per row	4
PE per chip	16
External data memory	8K bytes per PE
Technology	2·0 μm *n*-well CMOS (MOSIS)
Machine cycle	4-clock cycles
Clock frequency	20 MHz
Power consumption	810 mW

Table 2. DF-RISC-A processor specifications.

(see Table 2). In the current prototype, special hardware for multiplication and division have been designed. To obtain higher speed performance for some applications, these instructions and their associated hardware can be removed and arithmetic operations can be executed in software. This will increase the size of the DFGs but will improve the clocking speed. The architecture was modelled in VHDL to tune and to study the performance. The gate level implementation was made with the help of OASIS Silicon Compiler (Kozminski 1992) by translating the VHDL model to Logic3 hardware language. The layout of the prototype PE requires only 4261 gates and uses an area of 7512 × 8081 μm^2. This area is when a hardware multiplier (Sait *et al.* 1995) and divider are included.

Most of the dataflow computers that have been built to date were designed for complex operations; therefore it is not fair to compare these processors with this low complexity processor. However, the PE and array proposed by Koren *et al.* (1983) and Weiss *et al.* (1993) is similar to the one reported in this paper, but with enhancements explained in the following paragraphs.

The instruction format suggested by Koren *et al.* (1988) and Weiss *et al* (1993) uses undecoded bits for operand selection. This approach is very attractive, because it does not require decoding. However, it is very difficult to implement. Moreover, it is very difficult to differentiate between the first and second operands, and a maximum of two operands can be specified, not including the destination operand. This approach also increases the instruction length.

The design of Koren *et al.* (1988) and Weiss *et al.* (1993) is based on pure static architecture. The addition of rows of PEs to increase the array size will result in a bottleneck on the host communication bus. In the proposed design, which includes features of dynamic dataflow, the array is controlled by a GNC (§6). This combination of PEs and GNC makes a module, and several such modules (with minor hardware changes) can be combined together to make a larger array. All these modules can communicate with each other and the host through the GNC as shown in Fig. 16. Furthermore, in this design the PE identification number is been assigned during the reconfiguration/initialization phase (using the PE ID#, configuration register). The PEs are addressed through a single select line, which is

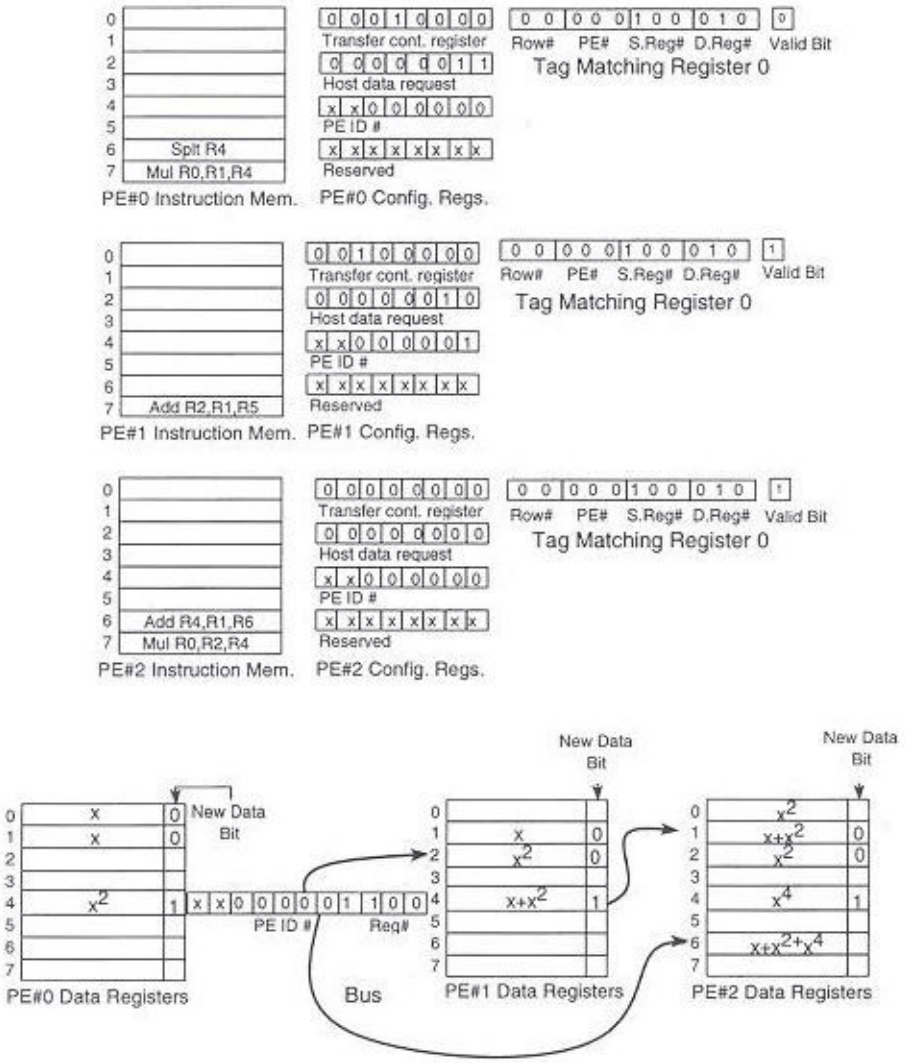

Figure 15. Instruction level mapping of the expression $x + x^2 + x^4$ on the PE array.

controlled by the GNC. This not only makes all the PEs identical, but also reduces the hardware to match the PE address. As explained earlier, it is also provides support for fault tolerance.

Owing to its high performance, ease of mapping and reconfigurability, this processor array can be used in a variety of applications. In particular, the dataflow graph of any arbitrary algorithm can be efficiently mapped for implementation onto this array of processors. Typical applications include array multiplication, addition, Fibonacci number generation and digital filtering.

Modern synthesis methodologies and tools were employed in the design of architecture, logic and layout. The simulation of the PE array was conducted at a high level of abstraction (in VHDL), whereas simulation of the final layout was performed for the complete PE and and the GNC.

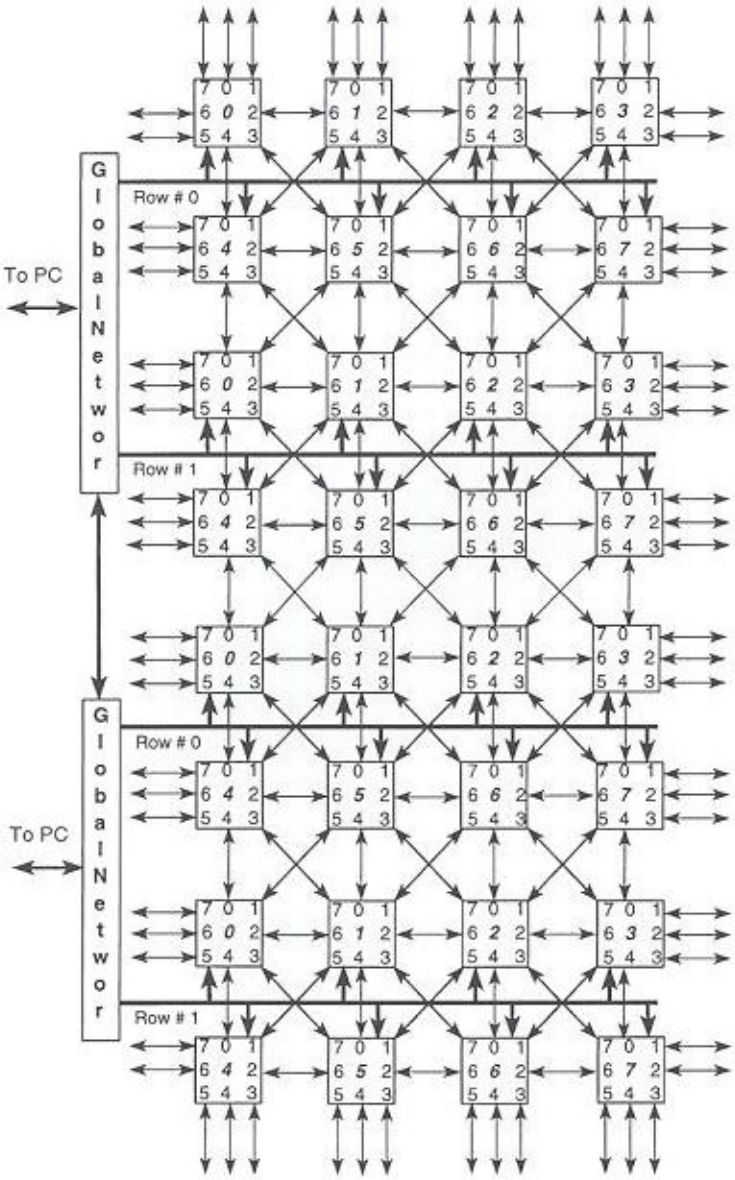

Figure 16. Interconnection of two modules.

ACKNOWLEDGMENTS

The authors thank King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia, for support under Project #COE/ARRAYS/177.

REFERENCES

ARVIND, and NIKHIL, R. S., 1989, Executing a program on the MIT tagged token dataflow architecture. *IEEE Transactions on Computers*, 1–29.

CAPPA, M., and HAMACHER, V. C., 1973, An augmented iterative array for high speed binary division. *IEEE Transactions on Computers*, **22**, 172–175.

DENNIS, J. B., 1973, *First version of a dataflow procedure language*. MAC Technical Memorandum, MIT, Cambridge, Massachusetts, U.S.A., p. 61.

DENNIS, J. B., LIM, W. Y.-P., and ACKERMAN, W. B., 1983, The MIT dataflow engineering model. *Proceedings of the IFIP 9th World Computer Conference*, pp. 553–560.

GAUDIOT, J., and BIC, L., 1991, *Advanced Topics in Dataflow Computing* (Prentice Hall).

GAUDIOT, J.-L., VEDDER, R. W., TUCKER, G. K., FINN, D., and CAMPBELL, M. L., 1985, A distributed VLSI architecture for efficient signal and data processing. *IEEE Transactions on Computers*, **34**, 1072–1087.

GRAFE, V. G., and HOCH, J. E., 1990, The epsilon-2 multiprocessor system. *Journal of Parallel and Distributed Processing*, **10**, 309–318.

GUILD, H. H., 1970, Some cellular logic arrays for non-restoring binary division. *Radio and Electronic Engineer*, **39**, 345–348.

GURD, J. R., KIRKHAM, C. C., and WATSON, I., 1985, The Manchester protype dataflow computer. *Communications of the Association for Computing Machinery*, **1**, 34–52.

KOREN, I., MENDELSON, B., PELED, I., and SILBERMAN, G. M., 1988, A data-driven VLSI array for arbitrary algorithms. *IEEE Computer*, **10**, 30–43.

KOZMINSKI, K., 1992, *OASIS: Open Architecture Silicon Implementation System Users Guide*. MCNC, Research Triangle Park, North Carolina, U.S.A.

KUNG, S. Y., LO, S. C., JEAN, S. N., and HWANG, L. N., 1987, Wavefront array processors concept to implementation. *IEEE Computer*, **7**, 18–33.

LEE, B., and HURSON, A. R., 1994, Dataflow architectures and multithreading. *IEEE Computer*, **27**, 27–39.

LIPSETT, R., SCHAEFER, C., and USSERY, C., 1987, *VHDL: Hardware Description and Design* (Kluwer Academic).

MAJITHIA, J. C., 1970, Nonrestoring binary division using a cellular array. *Electronic Letters*, **6**, 303–304.

MAYO, R. N., ARNOLD, M. H., SCOTT, W. S., STARK, D., AND HAMACHI, G. T., 1990, *DECWRL/ Livermore Magic Release*. Digital Western Research Laboratory.

NIKHIL, R. S., and ARVIND, 1989, Can dataflow subsume von Neumann computing. *Proceedings of the 16th International Symposium on Computer Architecture* (IEEE CS Press), pp. 262–272.

PAPADOPOULOS, G. M., 1988, *Implementation of a general purpose dataflow multiprocessor*. Technical Report, MIT Laboratory for Computer Science, Cambridge, Massachusetts, U.S.A. (TR-432).

SAIT, S. M., FAROOQUI, A. A., and BECKHOFF, G. F., 1995, A novel technique for fast multiplication *IEEE Phoenix Conference on Computers and Communication*.

TANENBAUM, A. S., 1990, *Structured Computer Organization* (Prentice-Hall).

WEISS, S., SPILLINGER, I. Y., and SILBERMAN, G. M., 1993, Architectural improvements for a data-driven VLSI processing array. *Journal of Parallel and Distributed Processing*, **19**, 308–322.

YUA, T., SHIMADA, T., HIRAKI, K., and KASHIWAGI, H., 1984, Sigma-1; a dataflow computer for scientific computations. *Proceedings of the 2nd International Conference on Vector and Parallel Processors in Computational Science*, pp. 28–31.